Android Development
Sqlite Usage

Key Concepts:

- **SQLite**

SQLite is an Open Source Database contained in Android. SQLite supports standard relational database features like SQL syntax, transactions and prepared statements. In addition it requires only little memory at runtime
SQLite is available on every Android device. Using an SQLite database in Android does not require any database setup or administration.

Relational database Definitions:

- A *relational database* is a collection of data organized in tables. There are relations among the tables. The tables are formally described. They consist of rows and columns.
- A *table* is a set of values that is organized using a model of vertical columns and horizontal rows. The columns are identified by their names.
- A database *row* represents a single, implicitly structured data item in a table. It is also called a tuple or a record.
- A *column* is a set of data values of a particular simple type, one for each row of the table. The columns provide the structure according to which the rows are composed.
- A *field* is a single item that exists at the intersection between one row and one column.
- A *primary key* uniquely identifies each record in the table.
- A *foreign key* is a referential constraint between two tables. The foreign key identifies a column or a set of columns in one (referencing) table that refers to a column or set of columns in another (referenced) table.
- A *schema* of a database system is its structure described in a formal language. It defines the tables, the fields, relationships, views, indexes, procedures, functions, queues, triggers and other elements.
- A *transaction* is an atomic unit of database operations against the data in one or more databases.
- An *index* is a data structure that improves the speed of data retrieval operations on a database table.

SQL (Structured Query Language)

- *SQL* (Structured Query Language) is a database computer language designed for managing data in relational database management systems.
- An SQL *result set* is a set of rows from a database returned by the SELECT statement. It also contains meta-information about the query such as the column names and the types and sizes of each column.
- The effects of all the SQL statements in a transaction can be either all committed to the database or all rolled back.

Reference: SQLite website: **http://www.sqlite.org**.

SQLite data definition language:
The *DDL* consists of SQL statements that define the database schema. The *schema* is the database structure described in a formal language. In relational databases, the schema defines the tables, views, indexes, relationships or triggers.

The SQLite supports the following DDL statements:
- CREATE
- ALTER TABLE
- DROP

- CREATE: In SQLite, the CREATE statement is used to create tables, indexes, views and triggers. To create a table, we give a name to a table and to its columns. Each column can have one of these data types:
  - NULL - The value is a NULL value
  - INTEGER - a signed integer
  - REAL - a floating point value
  - TEXT - a text string
  - BLOB - a blob of data

  Example:

  ```
  CREATE TABLE animaldata (Id INTEGER)
  ```

- ALTER: The ALTER TABLE statement changes the structure of the table.
  - SQLite supports a limited subset of the ALTER TABLE statement.

- This statement in SQLite allows a user to rename a table or to add a new column to an existing table.
- It is not possible to rename a column, remove a column, or add or remove constraints from a table.

Example:

```
ALTER TABLE animaldata ADD COLUMN area TEXT    // specifies type of column
```

- DROP: The DROP statement removes tables, indexes, views and triggers.

Example:

```
DROP TABLE IF EXISTS animaldata
```

Sqlite Data Manipulation Language (DML) allows manipulation of the data in the tables

INSERT: The INSERT statement is used to insert data into tables.

Example:

```
INSERT INTO animaldata (name, location) VALUES (Tiger, 'Tiger Trail');
```

DELETE: The DELETE keyword is used to delete data from tables.

Example:

```
DELETE FROM animaldata WHERE _id=1;
```

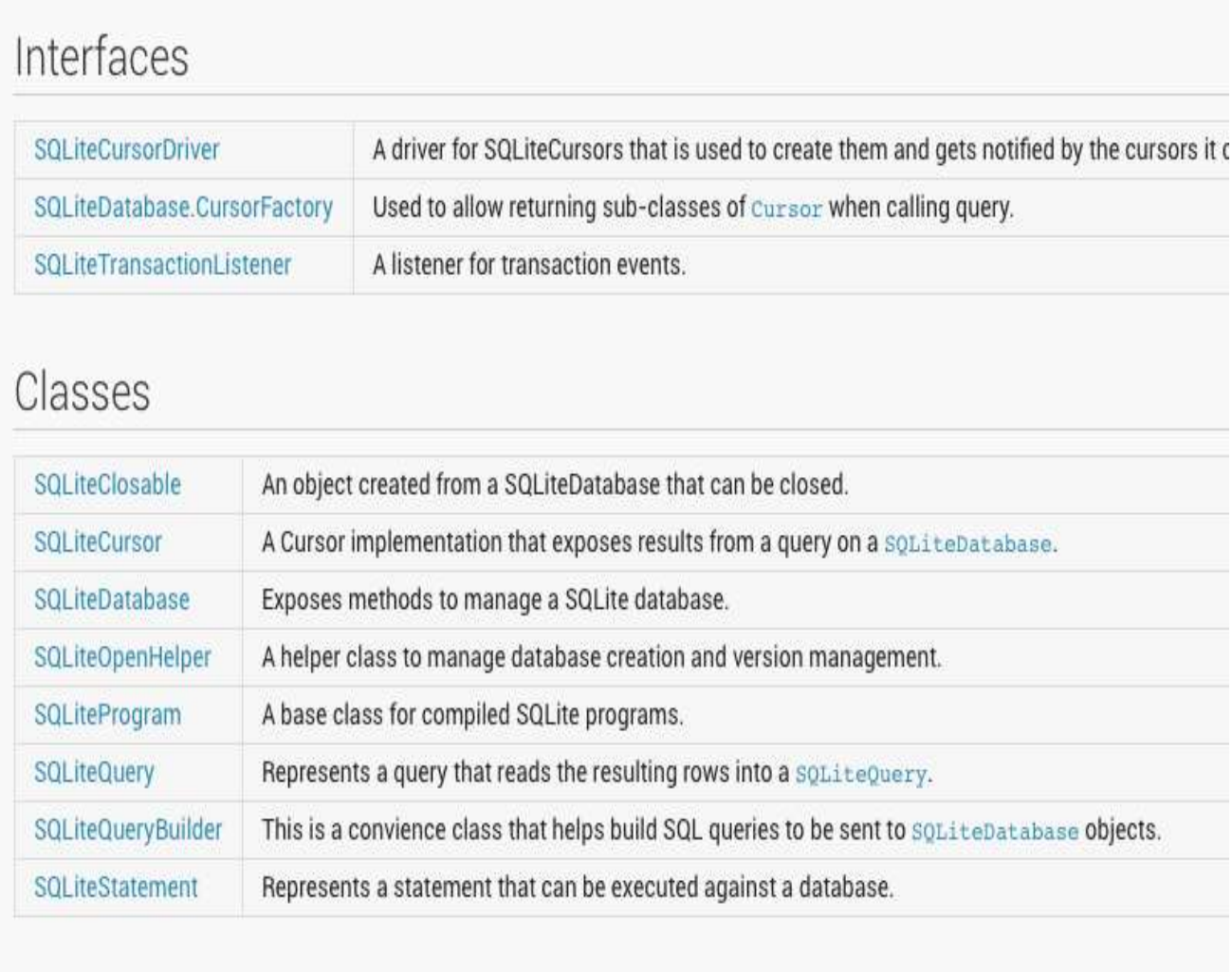
UPDATE: The UPDATE statement is used to change the value of columns in selected rows of a table.

Example:

```
UPDATE animaldata SET location ='Children Zoo' WHERE _id =1
```

Android SQLite Components/Concepts:

The package android.database contains all general classes for working with databases. android.database.sqlite contains the SQLite specific classes.



Interfaces

SQLiteCursorDriver	A driver for SQLiteCursors that is used to create them and gets notified by the cursors it c
SQLiteDatabase.CursorFactory	Used to allow returning sub-classes of `Cursor` when calling query.
SQLiteTransactionListener	A listener for transaction events.

Classes

SQLiteClosable	An object created from a SQLiteDatabase that can be closed.
SQLiteCursor	A Cursor implementation that exposes results from a query on a `SQLiteDatabase`.
SQLiteDatabase	Exposes methods to manage a SQLite database.
SQLiteOpenHelper	A helper class to manage database creation and version management.
SQLiteProgram	A base class for compiled SQLite programs.
SQLiteQuery	Represents a query that reads the resulting rows into a `SQLiteQuery`.
SQLiteQueryBuilder	This is a convience class that helps build SQL queries to be sent to `SQLiteDatabase` objects.
SQLiteStatement	Represents a statement that can be executed against a database.

http://developer.android.com/reference/android/database/sqlite/package-summary.html

SQLiteOpenHelper:

SQLiteOpenHelper contains methods to manage database creation and version management.
These methods include:

- onCreate: called when the database is created for the first time

- If your application creates a database, this database is by default saved in the directory Environment.getDataDirectory()/data/APP_NAME/databases/FILENAME.
- onUpgrade: called when the database needs to be upgraded
- getReadableDatabase(): create and/or open a database
- getWriteableDatabase(): create and/or open a database that will be used for reading and writing.

  getReadableDatabase() and getWriteableDatabase() give you access to an SQLiteDatabase object; either in read or write mode.

SQLiteDatabase:

The SQLiteDatabase class provides methods to interact with the database. SQLiteDatabase has methods to create, delete, execute SQL commands, and perform other common database management tasks.

These methods include

- Query: Queries the given table, returning a Cursor over the result set.
- Insert: Insert row into database
- Delete: delete rows in the database.
- Update: update row in the database.
- execSQL() : used to execute an SQL statement directly. The statement must NOT be a SELECT /INSERT/UPDATE/DELETE. (eg not anything that returns data)
- compileStatement: Compiles an SQL statement into a reusable pre-compiled statement object.
  - You may put ?s in the statement and fill in those values each time you want to run the statement.
  - SQLiteProgram.bindString can be used to fill in the statement.

SQLiteStatement:

The SQLiteStatement class represents statements to be executed on the database.

- executeInsert: Execute this SQL statement and return the ID of the row inserted due to this call.

- The SQL statement should be an INSERT for this to be a useful call.

Using a SQLite Database in your app.

- Determine the information you want to keep in your database and how you want to structure the database.
  - Determine if you need 1 or more than 1 table.
  - Determine what columns (information categories) you will have in the table
  - The database tables should use the identifier _id for the primary key of the table. Several Android functions rely on this standard.
- Create a class to handle the database activities.
  - In the exercise, this class is called DBHelper
  - In the constructor of this class, get an instance of your database.
  - Create the methods you need to insert and delete information into the database.
  - Construct methods to query for specific information in the database.
  - Some of the android SQLite methods can throw exceptions. Make sure to check documentation to determine if exceptions can be thrown & use try/catch routines.

- Create a class to handle creating and updating the database. This class extends from SQLiteOpenHelper.
  - In its constructor, call the super method of SQLiteOpenHelper, specifying database name and current database version.
  - Override onCreate (SQLiteDatabase db)
    - onCreate is called by the android system if the database does not exist.
    - Parameter is an SQLiteDatabase object that represents the database
  - Override onUpgrade (SQLiteDatabase db)
    - onUpgrade is called in the database version passed in the super method is higher than the one associated with your existing database. This allows you to update your database schema.
    - Multiple ways to update a schema including:
      - Drop the table and then recreate it
        - DROP TABLE IF EXISTS
      - Add a column to the end of the existing table
        - ALTER TABLE test1 ADD COLUMN bar TEXT; // specifies type of column
    - Parameter is an SQLiteDatabase object that represents the database

- This class can be an inner class inside your class that interacts with the database. In the exercise, this class is OpenHelper.

Exercise Objectives:

- Learn how to setup a SQLite database
- Learn to insert & delete information into the database
- Learn to query the database for specific information.

In this exercise, we are using Sqlite to develop a database system. This exercise extends the previous exercise to save the information in a sqlite database.

Note: Access to an SQLite database involves accessing the filesystem. This can be slow. Therefore it is recommended to perform database operations asynchronously, for example inside the AsyncTask class if your database will contain lots of data.

However, to simplify the example for this class, we will do the exercise on the UI thread. Since we have little data, it will not affect performance.

Step 1:

Make a backup copy of your previous exercise. If you do not have working source, you can use the helper files that are available in the resource area for the class. You will need to create your own application, but can copy/paste the code from the starter files for your layout and values xml files and animal.java file. You are missing the manifest file and 2 java files so you will have to handle those items. (If you need additional assistance, please contact the instructor).

Step 2:

Your UI is the same as the previous exercise and looks like the following:

Only java files with changes are mentioned in the following instructions. Changes in existing files are in bold.

Step 3: Animal.java file additions.

Add a long variable id. Add a “getter” and a “setter”

public class Animal {

```
    public final static String MAMMAL = "mammal";
    public final static String BIRD = "bird";
    public final static String REPTILE = "reptile";

    protected String name="";
    protected String location = "";
    protected String type="";
    protected long id = 0;

    public long getId() {
        return(id);
    }
```

```
    public void setId(long id) {
        this.id=id;
    }

    public String getName() {
        return(name);
    }

  … (rest of existing routine)
```

Step 4: Create your DBHelper.java file. (right click on java/<your package name>, New, class, call it DBHelper)

Create the code to create/update and get access to the database.
Add in the following code to the file. Do not remove your package name from the file the wizard created or your app will not build. When you copy the following you will see some items in red in Android Studio. You will resolve these in following steps. Make sure the spacing in the statement declaration for CREATE_TABLE is the same as shown below and that all quotes are straight and not curly. There is also a copy of this file in the resources section of canvas. You can copy sections of it if you'd like, but do not copy the package namespace.

```
import java.util.ArrayList;
import java.util.List;

import android.content.Context;
import android.database.Cursor;
import android.database.sqlite.SQLiteDatabase;
import android.database.sqlite.SQLiteOpenHelper;
import android.database.sqlite.SQLiteStatement;
import android.util.Log;

public final class DBHelper
{
   private static final String LOGTAG = "DBHelper";

   private static final String DATABASE_NAME    = "animal.db";
   private static final int    DATABASE_VERSION = 1;
   private static final String TABLE_NAME       = "animaldata";

   // Column Names
   public static final String     KEY_ID              = "_id";
```

```
 public static final String     KEY_NAME         = "name";
 public static final String     KEY_LOCATION     = "location";
 public static final String     KEY_TYPE         = "type";

 // Column indexes
 public static final int KEY_ID                  = 0;
 public static final int KEY_NAME                = 1;
 public static final int KEY_LOCATION            = 2;
 public static final int KEY_TYPE                = 3;

 private Context                 context;
 private SQLiteDatabase      db;
 public DBHelper (Context context) throws Exception
 {
    this.context = context;
    try {
       OpenHelper openHelper = new OpenHelper(this.context);
      // Open a database for reading and writing
      db = openHelper.getWritableDatabase();

    } catch (Exception e) {
      Log.e(LOGTAG, " DBHelper constructor:  could not get database " + e);
      throw (e);
    }
 }

// Helper class for DB creation/update
// SQLiteOpenHelper provides getReadableDatabase() and getWriteableDatabase() methods
// to get access to an SQLiteDatabase object; either in read or write mode.
//      A key defined as INTEGER PRIMARY KEY will autoincrement.

 private static class OpenHelper extends SQLiteOpenHelper
 {
    private static final String LOGTAG = "OpenHelper";

    private static final String CREATE_TABLE =
       "CREATE TABLE " +
       TABLE_NAME +
       "(" + KEY_ID + " INTEGER PRIMARY KEY, " +
       KEY_NAME + " TEXT, " +
       KEY_LOCATION  + " TEXT, " +
       KEY_TYPE    + " TEXT);";
```

```
OpenHelper (Context context)
 {
    super(context, DATABASE_NAME, null, DATABASE_VERSION);
 }

 /**
  * Creates the tables.
  * This function is only run once or after every Clear Data
  */
 @Override
 public void onCreate (SQLiteDatabase db)
 {
    Log.d(LOGTAG, " onCreate");
    try {
       db.execSQL(CREATE_TABLE);
    } catch (Exception e ) {
       Log.e(LOGTAG, " onCreate:  Could not create SQL database: " + e);
    }
 }

 /**
  * called, if the database version is increased in your application code.
  * This method updating an existing database schema or dropping the existing database
  * and recreating it via the onCreate() method.
  */

 @Override
 public void onUpgrade (SQLiteDatabase db, int oldVersion, int newVersion)
 {
    Log.w(LOGTAG,"Upgrading database, this will drop tables and recreate.");
    try {
       db.execSQL("DROP TABLE IF EXISTS " + TABLE_NAME);
       onCreate(db);
    } catch (Exception e ) {
       Log.e(LOGTAG, " onUpgrade:  Could not update SQL database: " + e);
    }

    // Technique to add a column rather than recreate the tables.
   //   String upgradeQuery_ADD_AREA =
   //       "ALTER TABLE "+ TABLE_NAME + " ADD COLUMN " + KEY_AREA + " TEXT ";
   //   if(oldVersion <2 ){
   //       db.execSQL(upgradeQuery_ADD_AREA);
```

```
    }
  }
}
```

Step 5: Set up your columns names and indexes

```
public final class DBHelper
{
   private static final String LOGTAG = "DBHelper";

   private static final String DATABASE_NAME    = "animal.db";
   private static final int    DATABASE_VERSION = 1;
   private static final String TABLE_NAME       = "animaldata";

   // Column Names
   public static final String     KEY_ID              = "_id";
   public static final String     KEY_NAME            = "name";
   public static final String     KEY_LOCATION        = "location";
   public static final String     KEY_TYPE            = "type";

   // Column indexes
   public static final int COLUMN_ID                  = 0;
   public static final int COLUMN_NAME                = 1;
   public static final int COLUMN_LOCATION            = 2;
   public static final int COLUMN_TYPE                = 3;

   private Context          context;
   private SQLiteDatabase   db;
```

Step 6: Insert. Set up statements to allow inserting of data into the database. These statements and method go inside the DBHelper class.

```
public final class DBHelper
{
   private static final String LOGTAG = "DBHelper";

   private static final String DATABASE_NAME    = "animal.db";
   private static final int    DATABASE_VERSION = 1;
   private static final String TABLE_NAME       = "animaldata";
```

```
// Column Names
public static final String      KEY_ID          = "_id";
public static final String      KEY_NAME        = "name";
public static final String      KEY_LOCATION    = "location";
public static final String      KEY_TYPE        = "type";

// Column indexes
public static final int COLUMN_ID       = 0;
public static final int COLUMN_NAME     = 1;
public static final int COLUMN_LOCATION = 2;
public static final int COLUMN_TYPE     = 3;

private Context          context;
private SQLiteDatabase   db;
private SQLiteStatement  insertStmt;

private static final String INSERT =
   "INSERT INTO " + TABLE_NAME + "(" +
   KEY_NAME + ", " +
   KEY_LOCATION + ", " +
   KEY_TYPE + ") values (?, ?, ?)";

public DBHelper (Context context) throws Exception
{
   this.context = context;
   try {
     OpenHelper openHelper = new OpenHelper(this.context);
     // Open a database for reading and writing
     db = openHelper.getWritableDatabase();
     //  compile a sqlite insert statement into re-usable statement object.
     insertStmt = db.compileStatement(INSERT);
   } catch (Exception e) {
     Log.e(LOGTAG, " DBHelper constructor:  could not get database " + e);
     throw (e);
   }
}

public long insert (Animal animalInfo)
{
   // bind values to the pre-compiled SQL statement "inserStmt"
   insertStmt.bindString(COLUMN_NAME, animalInfo.getName());
   insertStmt.bindString(COLUMN_LOCATION, animalInfo.getLocation());
   insertStmt.bindString(COLUMN_TYPE, animalInfo.getType());
```

```
    long value =-1;
    try {
      //  Execute the sqlite statement.
      value = insertStmt.executeInsert();
    } catch (Exception e) {
      Log.e(LOGTAG, " executeInsert problem: " + e);
    }
         Log.d (LOGTAG, "value="+value);
    return value;
  }
```

Step 7: Delete. Set up statements in your DBHelper to allow deletion of data into the database.

```
  public void deleteAll()
  {
     db.delete(TABLE_NAME, null, null);
  }

  // delete a row in the database
  public boolean deleteRecord(long rowId)
  {
   return db.delete(TABLE_NAME, KEY_ID + "=" + rowId, null) > 0;
  }
```

Step 8: Add the code to retrieve information from the database into DBHelper

```
  // Creates a list of animal info retrieved from the sqlite database.
   public List<Animal> selectAll()
   {
      List<Animal> list = new ArrayList<Animal>();


      // query takes the following parameters
      // dbName :  the table name
      // columnNames:  a list of which table columns to return
      // whereClause:  filter of selection of data;  null selects all data
      // selectionArg: values to fill in the ? if any are in the whereClause
      // group by:   Filter specifying how to group rows, null means no grouping
      // having:  filter for groups, null means none
      // orderBy:  Table columsn used to order the data, null means no order.

     // A Cursor provides read-write access to the result set returned by a database query.
```

```
    // A Cursor represents the result of the query and points to one row of the query result.
    Cursor cursor = db.query(TABLE_NAME,
        new String[] { KEY_ID, KEY_NAME, KEY_LOCATION, KEY_TYPE },
        null, null, null, null, null);


    if (cursor.moveToFirst())
    {
        do
        {
            Animal animalInfo = new Animal();
            animalInfo.setName(cursor.getString(COLUMN_NAME));
            animalInfo.setLocation(cursor.getString(COLUMN_LOCATION));
            animalInfo.setType(cursor.getString(COLUMN_TYPE));
            animalInfo.setId(cursor.getLong(COLUMN_ID));
            list.add(animalInfo);
        }
        while (cursor.moveToNext());
    }
    if (cursor != null && !cursor.isClosed())
    {
        cursor.close();
    }
    return list;
}
```

The complete code for the DBhelper class is listed at the end of this exercise. Please refer to it if you have issues with getting the DBhelper class to build or run.

Step 9: Open ZooListActivityFragment

Modify the ZooListActivityFragment fragment to use the database. The code to access the **dbHelper** is in **bold**

Declare a DBHelper variable.

```
public static class ZooListActivityFragment extends Fragment {

    List<Animal> animals=new ArrayList<Animal>();
    ArrayAdapter<Animal> adapter=null;
    private DBHelper dbHelper = null;
```

In onActivityCreated, instantiate the DBHelper and call the DBHelper selectAll method to get a list of all the animals in the database.

```
public void onActivityCreated(Bundle savedInstanceState){
    super.onActivityCreated(savedInstanceState);
    try {
        dbHelper = new DBHelper(getActivity());
        animals = dbHelper.selectAll();
    } catch (Exception e) {
        Log.d(TAG, "onCreate: DBHelper threw exception : " + e);
        e.printStackTrace();
    }

   ListView list=(ListView)getActivity().findViewById(R.id.zoo_animals);
   adapter=new AnimalAdapter(getActivity(),
        R.layout.row,
        animals);
```

In onSave, add the animal to the database when the animal is added to the list.

```
private void onSave(){
    Animal animal=new Animal();
    EditText name=(EditText)getActivity().findViewById(R.id.zoo_name);
        Spinner zoo_area = (Spinner) getActivity().findViewById(R.id.zoo_location);

    animal.setName(name.getText().toString());
    animal.setLocation(zoo_area.getSelectedItem().toString());

    RadioGroup types=(RadioGroup)getActivity().findViewById(R.id.zoo_animalType);

    switch (types.getCheckedRadioButtonId()) {
    case R.id.zoo_animalTypeMammal:
        animal.setType("mammal");
        break;
    case R.id.zoo_animalTypeBird:
        animal.setType("bird");
        break;
    case R.id.zoo_animalTypeReptile:
```

```
            animal.setType("reptile");
            break;
        }

            long animalId = 0;
            if (dbHelper != null) {
                animalId = dbHelper.insert(animal);
                animal.setId(animalId);
            }

        // Add the object at the end of the array.
        adapter.add(animal);
        // Notifies that the underlying data has changed, views reflecting the data should refresh.
        adapter.notifyDataSetChanged();

        // Remove the soft keyboard after hitting the save button
        InputMethodManager inputManager = (InputMethodManager)
                getActivity().getSystemService(Context.INPUT_METHOD_SERVICE);
        inputManager.hideSoftInputFromWindow(getActivity().getCurrentFocus().getWindowToken(),
                InputMethodManager.HIDE_NOT_ALWAYS);
    }
```

In onDelete, delete the database entry when the animal is deleted from the list.

```
    private void onDelete(View view, int position){
       // When clicked, delete the item that was clicked.
      // (Show a toast to indicate what is occurring)
        String item = "deleting: " +
            ((TextView)view.findViewById(R.id.row_name)).getText().toString();
        Toast.makeText(getActivity(), item, Toast.LENGTH_SHORT).show();

        Animal animal = adapter.getItem(position);
        if (animal != null) {
            Log.d(TAG, " onItemClick: " + animal.name);
        }

        // database delete record
        if (dbHelper != null) dbHelper.deleteRecord(animal.getId());
        // Removes the object from the array
        adapter.remove(animal);
        // Notifies that the underlying data has changed
        adapter.notifyDataSetChanged();
    }
```

Step 10:

Run your Android program. Add in some data. Close your app. Re-open the app. You should see that your app displays the saved data.

Appendix: DBHelper class

```
package com.fsavage.zoolist;

import java.util.ArrayList;
import java.util.List;
import android.content.Context;
import android.database.Cursor;
import android.database.sqlite.SQLiteDatabase;
import android.database.sqlite.SQLiteOpenHelper;
import android.database.sqlite.SQLiteStatement;
import android.util.Log;

public final class DBHelper
{
   private static final String LOGTAG = "DBHelper";

   private static final String DATABASE_NAME    = "animal.db";
   private static final int    DATABASE_VERSION = 1;
   private static final String TABLE_NAME      = "animaldata";

   // Column Names
   public static final String     KEY_ID              = "_id";
   public static final String     KEY_NAME            = "name";
   public static final String     KEY_LOCATION    = "location";
   public static final String     KEY_TYPE        = "type";

   // Column indexes
   public static final int COLUMN_ID              = 0;
   public static final int COLUMN_NAME            = 1;
   public static final int COLUMN_LOCATION      = 2;
   public static final int COLUMN_TYPE            = 3;

   private Context              context;
   private SQLiteDatabase     db;
   private SQLiteStatement    insertStmt;

   private static final String INSERT =
         "INSERT INTO " + TABLE_NAME + "(" +
               KEY_NAME + ", " +
               KEY_LOCATION + ", " +
               KEY_TYPE + ") values (?, ?, ?)";
```

```
  public DBHelper (Context context) throws Exception
  {
     this.context = context;
     try {
        OpenHelper openHelper = new OpenHelper(this.context);
// Open a database for reading and writing
        db = openHelper.getWritableDatabase();
//  compile a sqlite insert statement into re-usable statement object.
        insertStmt = db.compileStatement(INSERT);

     } catch (Exception e) {
        Log.e(LOGTAG, " DBHelper constructor:  could not get database " + e);
        throw (e);
     }
  }
  public long insert (Animal animalInfo)
  {
     // bind values to the pre-compiled SQL statement "inserStmt"
     insertStmt.bindString(COLUMN_NAME, animalInfo.getName());
     insertStmt.bindString(COLUMN_LOCATION, animalInfo.getLocation());
     insertStmt.bindString(COLUMN_TYPE, animalInfo.getType());

     long value =-1;
     try {      //  Execute the sqlite statement.
        value = insertStmt.executeInsert();
     } catch (Exception e) {
        Log.e(LOGTAG, " executeInsert problem: " + e);
     }
     Log.d (LOGTAG, "value="+value);
     return value;
  }

  public void deleteAll()
  {
     db.delete(TABLE_NAME, null, null);
  }

  // delete a row in the database
  public boolean deleteRecord(long rowId)
  {
     return db.delete(TABLE_NAME, KEY_ID + "=" + rowId, null) > 0;
  }

  public List<Animal> selectAll()
  {
     List<Animal> list = new ArrayList<Animal>();
```

```
  // query takes the following parameters
  // dbName :  the table name
  // columnNames:  a list of which table columns to return
  // whereClause:  filter of selection of data;  null selects all data
  // selectionArg: values to fill in the ? if any are in the whereClause
  // group by:   Filter specifying how to group rows, null means no grouping
  // having:  filter for groups, null means none
  // orderBy:  Table columsn used to order the data, null means no order.

  // A Cursor provides read-write access to the result set returned by a database query.
  // A Cursor represents the result of the query and points to one row of the query result.
  Cursor cursor = db.query(TABLE_NAME,
        new String[] { KEY_ID, KEY_NAME, KEY_LOCATION, KEY_TYPE },
        null, null, null, null, null);

 if (cursor.moveToFirst())
  {
     do
     {
        Animal animalInfo = new Animal();
        animalInfo.setName(cursor.getString(COLUMN_NAME));
        animalInfo.setLocation(cursor.getString(COLUMN_LOCATION));
        animalInfo.setType(cursor.getString(COLUMN_TYPE));
        animalInfo.setId(cursor.getLong(COLUMN_ID));
        list.add(animalInfo);
     }
     while (cursor.moveToNext());
  }
  if (cursor != null && !cursor.isClosed())
  {
     cursor.close();
  }
  return list;
}

// Helper class for DB creation/update
// SQLiteOpenHelper provides getReadableDatabase() and getWriteableDatabase() methods
// to get access to an SQLiteDatabase object; either in read or write mode.

private static class OpenHelper extends SQLiteOpenHelper {
  private static final String LOGTAG = "OpenHelper";

  private static final String CREATE_TABLE =
        "CREATE TABLE " +
             TABLE_NAME +
             "(" + KEY_ID + " INTEGER PRIMARY KEY, " +
             KEY_NAME + " TEXT, " +
             KEY_LOCATION + " TEXT, " +
```

```
            KEY_TYPE + " TEXT);";

    OpenHelper(Context context) {
      super(context, DATABASE_NAME, null, DATABASE_VERSION);
    }

    /**
     * Creates the tables.
     * This function is only run once or after every Clear Data
     */
    @Override
    public void onCreate(SQLiteDatabase db) {
      Log.d(LOGTAG, " onCreate");
      try {
        db.execSQL(CREATE_TABLE);
      } catch (Exception e) {
        Log.e(LOGTAG, " onCreate:  Could not create SQL database: " + e);
      }
    }

    /**
     * called, if the database version is increased in your application code.
     * This method updating an existing database schema or dropping the existing database
     * and recreating it via the onCreate() method.
     */

    @Override
    public void onUpgrade(SQLiteDatabase db, int oldVersion, int newVersion) {
      Log.w(LOGTAG, "Upgrading database, this will drop tables and recreate.");
      try {
        db.execSQL("DROP TABLE IF EXISTS " + TABLE_NAME);
        onCreate(db);
      } catch (Exception e) {
        Log.e(LOGTAG, " onUpgrade:  Could not update SQL database: " + e);
      }

      // Technique to add a column rather than recreate the tables.
      //   String upgradeQuery_ADD_AREA =
      //      "ALTER TABLE "+ TABLE_NAME + " ADD COLUMN " + KEY_AREA + " TEXT ";
      //   if(oldVersion <2 ){
      //      db.execSQL(upgradeQuery_ADD_AREA);
      }
    }
}
```